```
# USING PYTHON 3.6.7

# Homework Number: 6
# Name: Michael Cupka
# ECN Login: mcupka
# Due Date: February 26, 2019

import sys
import os
import numpy
from PrimeGenerator import PrimeGenerator
from BitVector import *

# given code to solve p root with necessary precision
def solve_pRoot(p,y):
    p = int(p);
    y = int(y);
    # Initial guess for xk
    try:
        xk = int(pow(y,1.0/p));
    except:
        # Necessary for larger value of y
        # Approximate y as 2^a * y0
        y0 = y;
        a = 0;
        while (y0 > sys.float_info.max):
            y0 = y0 >> 1;
            a += 1;
        # log xk = log2 y / p
        # log xk = (a + log2 y0) / p
        xk = int(pow(2.0, ( a + np.log2(float(y0)) )/ p ));

    # Solve for x using Newton's Method
    err_k = int(pow(xk,p))-y;
    while (abs(err_k) > 1):
        gk = p*int(pow(xk,p-1));
        err_k = int(pow(xk,p))-y;
        xk = int(-err_k/gk) + xk;
    return xk


# function to find the gcd of two integers using euclid's extended algorithm
def gcd_euclid(a: int, b: int) -> int:
    # this algorithm is copied from the lecture 5 slides
    while b:
        a, b = b, a%b
    return a


# function to generate n, and d given e. assuming 256 bit RSA (128 bit p and q)
def get_keys(e):
    pgen = PrimeGenerator(bits=128)

    pq_cond = False
    while not pq_cond:

        p, q = (pgen.findPrime(), pgen.findPrime())
        p_bv, q_bv = BitVector(size=128, intVal=p), BitVector(size=128, intVal=q)

        # now test to see if the conditions are met
        pq_cond = True
        # check that the first bits are set
```

```
        if p_bv[0] == 0 or q_bv[0] == 0: pq_cond = False
        # check that p and q are not equal
        if p == q: pq_cond = False
        # check that p-1 and q-1 are coprime to e
        if gcd_euclid(p-1, e) != 1 or gcd_euclid(q-1, e) != 1: pq_cond = False
        # if any of the above checks are not passed, a new p and q will be generated
and checked

    # calculate n value from p and q
    n = p * q

    # find d, the MI of e in mod n
    e_bv = BitVector(intVal=e)
    d = e_bv.multiplicative_inverse(BitVector(intVal=n)).int_val()
    return e, d, n

def rsa_enc(key_tuple, input_fname, output_fname):
    e, d, n = key_tuple
    input_bv = BitVector(filename=input_fname)
    output_file = open(output_fname, 'w')
    while input_bv.more_to_read:
        one_block_bv = input_bv.read_bits_from_file(128)
        one_block_bv.pad_from_right(128 - one_block_bv.length()) # pad the final
block from the right
        one_block_bv.pad_from_left(128) # pad each block from the left to make 256
bits

        # encrypt the block
        enc_block_bv = BitVector(intVal=pow(one_block_bv.int_val(), e, n), size = 256)

        # write the block to the output file in hex format
        hexstr = enc_block_bv.get_hex_string_from_bitvector()
        output_file.write(hexstr)


# function to utilize CRT to get M^3 mod n
def get_m_cubed(C_vals, n, k1, k2, k3):
    # find M^3 (mod n), using factors k1, k2, and k3 of n and 3 C values

    # this code follows the procedure in the lecture slides
    K1 = k2 * k3
    K2 = k1 * k3
    K3 = k1 * k2

    bv_K1 = BitVector(intVal=K1)
    bv_K2 = BitVector(intVal=K2)
    bv_K3 = BitVector(intVal=K3)


    K1_inv = bv_K1.multiplicative_inverse(BitVector(intVal= k1)).int_val()
    K2_inv = bv_K2.multiplicative_inverse(BitVector(intVal= k2)).int_val()
    K3_inv = bv_K3.multiplicative_inverse(BitVector(intVal= k3)).int_val()

    result = (C_vals[0] * K1 * K1_inv + C_vals[1] * K2 * K2_inv + C_vals[2] * K3 *
K3_inv) % n

    return result

def crack_rsa(filename1, filename2, filename3, n1, n2, n3, output_fname):
    file1_bv = BitVector(filename=filename1)
    file2_bv = BitVector(filename=filename2)
    file3_bv = BitVector(filename=filename3)
```

```
    files = [file1_bv, file2_bv, file3_bv]
    output_file = open(output_fname, 'wb')

    C_vals = [None, None, None]
    N = n1 * n2 * n3

    # for each block, get M^3 mod N, then solve for M
    while file1_bv.more_to_read:
        # get the c val for each
        for index, file in enumerate(files):
            one_block_hex_bv = file.read_bits_from_file(512)
            one_block_bv =
BitVector(hexstring=one_block_hex_bv.get_bitvector_in_ascii())
            C_vals[index] = one_block_bv.int_val()

        # get M^3 mod N
        Mcubed = get_m_cubed(C_vals, N, n1, n2, n3)

        # solve for M
        M_bv = BitVector(intVal=solve_pRoot(3, Mcubed), size=128)
        # output cracked text to file
        M_bv.write_to_file(output_file)




if __name__ == '__main__':
    # generate 3 public and private keys
    keys_list = []
    for _ in range(3):
        keys_list.append(get_keys(3))

    # encrypt the plaintext using each key
    for index, key in enumerate(keys_list):
        rsa_enc(key, sys.argv[1], 'enc' + str(index + 1) + '.txt')

    # use the files to crack the encrypted text
    crack_rsa('enc1.txt', 'enc2.txt', 'enc3.txt', keys_list[0][2], keys_list[1][2],
keys_list[2][2], sys.argv[2])

    # save the n values used for turning in on the hard-copy submission
    n_file = open('n_1.txt', 'w')
    n_file.write(str(keys_list[0][2]))
    n_file.close()
    n_file = open('n_2.txt', 'w')
    n_file.write(str(keys_list[1][2]))
    n_file.close()
    n_file = open('n_3.txt', 'w')
    n_file.write(str(keys_list[2][2]))
    n_file.close()
```